13 OpenGym AI

Lab Objective: *OpenGym AI is a module designed to learn and apply reinforcement learning. The purpose of this lab is to learn the variety of functionalities available in OpenGym AI and to implement them in various environments. Applying reinforcement learning techniques into OpenGym AI will take place in future labs.*

OpenGym AI is a module used to perform reinforcement learning. It contains a collection of environments where reinforcement learning can be used to accomplish various tasks. These environments include performing computer functions such as copy and paste, playing Atari video games, and controlling robots. To install OpenGym AI, run the following code.

```
>>> pip install gym
```

Environments

Each environment in OpenGym AI can be thought of as a different scenario where reinforcement learning can be applied. A catalog of all available environments can be found using the following code:

```
>>> from gym import envs
>>> print(envs.registry.all())
dict_values([EnvSpec(Copy-v0), EnvSpec(RepeatCopy-v0), EnvSpec(ReversedAddition↩
    -v0), EnvSpec(ReversedAddition3-v0), EnvSpec(DuplicatedInput-v0), EnvSpec(↩
    Reverse-v0), EnvSpec(CartPole-v0),...
```

Note that some of these environments require additional software. To learn more about each environment, its scenario and necessary software, visit `gym.openai.com/envs`.

To begin working in an environment, identify the name of the environment and initialize the environment. Once intialized, make sure to reset the environment. Resetting the environment is necessary to begin using the environment by putting everything in the correct starting position. For example, the environment `"NChain-v0"` presents a scenario where a player is traversing a chain with n states. Restarting the environment puts the player at the beginning of the chain. Once the environment is complete, make sure to close the environment. Closing the environment tells the computer to stop running the environment as to not run the environment in the background.

```
>>> import gym

>>> # Get NChain-v0 environment
env = gym.make('NChain-v0')

>>> # Reset the environment
>>> env.reset()
0

>>> # Close the environment
>>> env.close()
```

Action Space

Once reset, the player in the environment can then perform actions from the action space. In "NChain-v0", the action space has 2 actions; move forward one state or return to the beginning of the chain. To perform an action, use the function `step`, which accepts the action as a parameter and returns an observation (more on those later). If the action space is discrete, then actions are defined as integers 0 through n, where n is the number of actions. The action each integer represents can be found in the documentation of each environment.

```
>>> # Determine the number of actions available
>>> env.action_space
Discrete(2)

>>> # Reset environment and perform a random action
>>> env.step(env.action_space.sample())
(1, 0, False, {})
```

However, not all action spaces are discrete. Consider the environment "GuessingGame-v0". The purpose of this game is to guess within 1% of a random number in the interval $[-1000, 1000]$ in 200 guesses. Since the number is not required to be an integer, and each action is guessing a number, it does not make sense for the action space to be discrete. Rather this action space should be an interval In OpenGym AI, this action space is described as an n-dimensional array `Box(n,)`. This means an feasible action is an n-dimensional vector. To identify the range of the box, use the attributes `high` and `low`. Thus, in the environment "GuessingGame-v0", the action space will be a 1-dimensional box with range $[-1000, 1000]$.

```
>>> # Initialize Guessing Game environment
>>> env = gym.make("GuessingGame-v0")
>>> observation = env.reset()
>>> # Check size of action space
>>> env.action_space
Box(1,)

>>> # Check range of action space
>>> env.action_space.high
```

```
array([10000.], dtype=float32)

>>> env.action_space.low
array([-10000.], dtype=float32)
```

Observation Space

The observation space contains all possible observations given an action. For example, in `"NChain-v0"`, an observation would be the position of the player on the chain and in `"GuessingGame-v0"`, the observation would be whether the guess is higher than, equal to, or lower than the target. The observation from each action can be found in the tuple returned by `step`. This tuple tells us the following information:

1. `observation`: The current state of the environment. For example, in `"GuessingGame-v0"`, 0 indicates the guess is too high, 1 indicates the guess is on target, and 2 indicates the guess is too low.
2. `reward`: The reward given from the observation. In most environments, maximizing the total reward increases performance. For example, the reward for each observation is 0 in `"GuessingGame-v0"` unless the observation is within 1% of the target.
3. `done`: A boolean indicating if the observation terminates the environment.
4. `info`: Various information that may be helpful when debugging.

Consider the code below.

```
>>> env = gym.make("GuessingGame-v0")
>>> observation = env.reset()

>>> # Make a random guess
>>> env.step(env.action_space.sample())
(1, 0, False, {'guesses': 1, 'number': 524.50509074})
```

This tuple can be interpreted as follows:

1. The guess was too high.
2. The guess was not with 1% of the target.
3. The environment is not terminated.
4. Information that may help debugging (the number of guesses made so far and the target number).

Problem 1. The game Blackjack[a] is a card game where the player receives two cards from a facecard deck. The goal of the player is to get cards whose sum is as close to 21 as possible without exceeding 21. In this version of Blackjack, an ace is considered 1 or 11 and any facecard is considered 10. At each turn, the player may choose to take another card or stop drawing cards. If their card sum does not exceed 21, they may take another card. If it does, they lose. After the player stops drawing cards, the computer may play the same game. If the computer gets closer to 21 than the player, the player loses.

The environment `"Blackjack-v0"` is an OpenGym AI environment that plays blackjack. The actions in the action space are 0 to stop drawing and 1 to draw another card. The observation (first entry in the tuple returned by `step`) is a tuple containing the total sum of the players hand, the first card of the computer's hand, and whether the player has an ace. The reward (second entry in the tuple returned by `step`) is 1 if the player wins, -1 if the player loses, and 0 if there is a draw.

Write a function `random_blackjack()` that accepts an integer n. Initialize `"Blackjack-v0"` n times and each time take random actions until the game is terminated. Return the percentage of games the player wins. Use your function to print the win percentage after 500 games.

[a]For more on how to play Blackjack, see `https://en.wikipedia.org/wiki/Blackjack`.

Understanding Environments

Because each action and observation space is made up of numbers, good documentation is imperative to understanding any given environment. Fortunately, most environments in OpenAI Gym are very well documented. Documentation for any given environment can be found through `gym.openai.com/envs` by clicking on the github link in the environment.

Most documentation follows the same pattern. There is a docstring which includes a description of the environment, a detailed action space, a detailed observation space, and explanation of rewards. It is always helpful to refer to this documentation when working in an OpenGym AI environment.

In addition to documentation, certain environments can be understood better through visualization. For example, the environment `"Acrobot-v1"` displays an inverted pendulum. Visualizing the environment allows the user to see the movement of the inverted pendulum as forces are applied to it. This can be done with the function `render()`. When using `render()`, ALWAYS use a try-finally block to close the environment. This ensures that the video rendering ends no matter what.

```
>>> # Get environment
>>> env = gym.make("Acrobot-v1")

>>> # Take random actions and visualize each action
>>> try:
>>>     env.reset()
>>>     done = False
>>>     while not done:
>>>        env.render()
>>>        obs, reward, done, info = env.step(env.action_space.sample())
>>>        if done:
>>>            break
```

```
>>> finally:
>>>     env.close()
```

Figure 13.1: Rendering of "Acrobot-v1"

Problem 2. Write a function `blackjack()` which runs a naive algorithm to win blackjack. The function should receive an integer `n`. If the player's hand is less than or equal to `n`, the player should draw another card. If the player's hand is more than `n`, they should stop playing. Within the function, run the algorithm 10000 times and return the percentage of games the player wins.

For $n = 1, 2, \ldots, 21$, plot the average win rate returned by your function. Identify which value(s) of n wins most often.

Solving An Environment

One way to solve an environment is to use information from the current observation to choose our next action. For example, consider "GuessingGame-v0". Each observation tells us whether the guess was too high or too low. After each observation, the interval where the target lies continues to get smaller. By choosing the midpoint of the current interval where the target lies, the true target can be identified much faster.

Problem 3. The environment "CartPole-v0" presents a cart with a vertical pole. The goal of the environment is to keep the pole vertical as long as possible. Write a function `cartpole()` which initializes the environment and keeps the pole vertical as long as possible based on the velocity of the tip of the pole. Return the number of steps it takes before it terminates. The number of steps should be at least 120 and on average about 180.

Run the game a single time and render the environment at each step. Then without rendering, run your function 100 times and print the average number of steps before it terminates.

(Hint: Use the documentation of the environment to determine the meaning of each action and observation. It can be found at `https://github.com/openai/gym/wiki/CartPole-v0`.)

Problem 4. The environment `"MountainCar-v0"` shows a car in a valley. The goal of the environment is to get the car to the top of the right mountain. The car can be driven forward (toward the goal) with the action 2, can be driven backward with the action 0, and will be put in neutral with the action 1. Note that the car cannot immediately get up the hill because of gravity. In order to move the car to goal, momentum will need to be gained by going back and forth between both sides of the valley. Each observation is a 2-dimensional array, containing the x position and the velocity of the car respectively. Using the given position and velocity of the car, write a function `car()` that solves the `"MountainCar-v0"` environment. Return the number of steps it takes before it terminates. The number of steps should be less than 180.

Run the game a single time and render the environment at each step. Then without rendering, run your function 100 times and print the average number of steps before it terminates.

Q-Learning

While naive methods like the ones above can be useful, reinforcement is a much better approach for using OpenAI Gym. Reinforcement learning is a subfield of machine learning where a problem is attempted over and over again. Each time a method is used to solve the problem, the method adapts based on the information gained from the previous attempt. Information can be gained from the sequence of observations and the total reward earned.

A simple reinforcement method is called *Q-learning*. While the details of Q-learning will not be explained in detail, the main idea is that the next action is not only based on the reward of the current action, but also of the next action. Q-learning creates a Q-table, which is an $n \times m$ dimensional array, where n is the number of observations and m is the number of actions. For each state, the optimal action is the action that maximizes the value in the Q-table. In other words, if I am at observation i, the best action is the argmax of row i in the Q-table.

Q-learning requires 3 hyperparameters:

1. `alpha`: the learning rate. This determines whether to accept new values into the q-table.
2. `gamma`: the discount factor. The discount factor determines how important the reward of the current action is compared to the following action.
3. `epsilon`: the maximum value. This is the max reward that can be earned from a future action (not the current).

These hyperparameters can be changed to create different Q-tables. The following function will generate the optimal Q-table.

```
def find_qvalues(env,alpha=.1,gamma=.6,epsilon=.1):
    """
    Use the Q-learning algorithm to find qvalues.
    Parameters:
        env (str): environment name
        alpha (float): learning rate
```

```
        gamma (float): discount factor
        epsilon (float): maximum value
    Returns:
        q_table (ndarray nxm)
    """
    # Make environment
    env = gym.make(env)
    # Make Q-table
    q_table = np.zeros((env.observation_space.n,env.action_space.n))

    # Train
    for i in range(1,100001):
        # Reset state
        state = env.reset()

        epochs, penalties, reward, = 0,0,0
        done = False

        while not done:
            # Accept based on alpha
            if random.uniform(0,1) < epsilon:
                action = env.action_space.sample()
            else:
                action = np.argmax(q_table[state])

            # Take action
            next_state, reward, done, info = env.step(action)

            # Calculate new qvalue
            old_value = q_table[state,action]
            next_max = np.max(q_table[next_state])

            new_value = (1-alpha) * old_value + alpha * (reward + gamma * ↩
                next_max)
            q_table[state, action] = new_value

            # Check if penalty is made
            if reward == -10:
                penalties += 1

            # Get next observation
            state = next_state
            epochs += 1

        # Print episode number
        if i % 100 == 0:
            clear_output(wait=True)
            print(f"Episode: {i}")
```

```
    print("Training finished.")
    return q_table
```

Problem 5. The environment `"Taxi-v3"` shows a taxi on a city grid. The goal of this environment is to pick up a passenger in a taxi and drop them off at their destination as fast as possible (see `https://gym.openai.com/envs/Taxi-v3/`). First, initialize and render the environment. Randomly act until the environment is done and calculate the total reward. Next, use `find_qvalues()` to get the optimal Q-table of the environment. Set `alpha=.1`, `gamma=.6`, and `epsilon=.1`. Render the environment, use the qtable to move through it, and calculate the total reward.

Finally, write a function `taxi()` which initializes the `"Taxi-v3"` environment (without rendering). Run each scenario 10000 times, and return the average random total reward and the average Q-learning total reward.

(Hint: Use the documentation found at `https://github.com/openai/gym/blob/master/gym/envs/toy_text/taxi.py` to understand the environment better).